Overview

The purpose of this document is to provide a brief introduction to the template language implemented in UMA. It assumes you have a working knowledge of the object/property metamodel of UMA.

It is not intended to provide an exhaustive explanation of the architecture. UMA team members can provide you with such an explanation and they can direct you to many code samples that use this facility.

The purpose of the UMA Template Language (UTL) is to provide scripting capabilities to any UMA-based application. These capabilities can be used for various features, including (but not limited to):

- **Macro expansion**: Runtime expansion of code templates against the model. Examples include trigger code and stored procedures in OR•Compass, and Visual Basic and Java components in SQL•Compass.
- **Scripting**: Execution of a series of commands against the model. An example is the controller code for Forward Engineering in OR•Compass.

The architecture of UTL is intended to promote its use in as many aspects of an UMA-based product as possible. This is accomplished by allowing the user[1] maximum flexibility in tailoring the language to the specific needs of his product, while providing as much implementation support as possible and exposing mechanisms that allow reuse of new implementation code.

To accomplish these goals, UTL:

1. defines a minimal syntax for the language;
2. allows the user to write procedures and interators;
3. provides a mechanism for registering the procedures and iterators in registries.

Syntax Summary

The following is a brief summary of the syntax of UTL intended to provide the bare minimum necessary to understand the implementation. More information can be found in the documentation shipped with OR•Compass 1.0.

Template code consists of literals, macros, comments and a few operator/scoping characters. Currently, UTL views all code as strings: numeric values are described by their string representation; most macros evaluate to strings. etc.[2]

---

[1] In this document, "user" refers to the UMA application developer, not the end-user of the application.
[2] UTL 1.5, expected this fall, will also support numeric modes of operation that will provide better performance when dealing with numbers. This will be useful for features such as volumetric calculations.

Literals

In template source code, all text outside of curly braces is treated as a literal, and is emitted exactly as typed. Within curly braces, text inside double quotes is treated as a literal. The former allows large blocks of boilerplate to be entered, e.g. the body of a stored procedure. The latter allows you to embed literals within macro calls.

Example:

```
This text would emit just like this.
{"So would this text."}
```

Macros

Macros are special instructions to the macro processor...basically procedures. All macros are contained within curly braces. Macros basically fall into two categories: procedure macros and iterator macros.

Procedure macros are designed to perform some work. They may expand to a value; they may declare a variable; they may invoke a process. The action(s) performed are entirely specified by the designer of the macro. The only constraint upon them is that they return True upon successful completion of their task and False upon failure.

Example—the following shows a string literal, followed by a macro call for getting the page number when printing:

```
My Model Report - Page {HeaderPage}     ⇦ Input
My Model Report - Page 1                ⇦ Output
```

Iterator macros allow the user to traverse across data structures. Iterators are distinguished by the keywords begin and end that delimit a block of code following the iterator declaration. All of the code within this begin/end block will be executed once for each iteration. When the iterator has moved across all objects in its pool, control will break out of the iteration block and execute the first statement after the block.

Example—the following executes the procedure macro DoSomething once for each element returned by the MyIterator macro. Note the curly braces surrounding the entire code fragment...indicating that all code within is to be treated as macro code.

```
{
   MyIterator
   begin
      DoSomething
   end
}
```

Macros, both procedure and iterator, can take parameters. Parameters are strings, or anything that evaluates to a string, such as another macro. If a macro takes parameters, they are enclosed in parentheses after the macro. Since forward declaration of macros is not required, macros may accept variable-length parameter lists, if desired.

Example:

```
{ MacroWithParameters("foo") }
```

Control blocks

Occasionally, it is desirable to have an entire block of template code fail if any piece of it fails. An example would be where there was some boilerplate text followed by an optional value; if the value was not present, the boilerplate should not be emitted.

Any code enclosed in square brackets is *conditional*: it will all fail if any piece of it fails. "Failure" means that a macro will not execute and a literal will not be emitted.

Example—if there is no middle name, the period will not emit:

```
{ FirstName [MiddleInitial "."] LastName }
```

Conditional blocks hide their internal scope. This means that the failure of a conditional block has no effect on the surrounding code. Any code enclosed in angle brackets is *propagating conditional*: it will fail if any piece of it fails and it will propagate that failure out to the next block.

The best way to distinguish these two types of blocks is with examples. In both examples, the Print macro emits the word "foo" and the Fail macro fails no matter what.

Example of conditional block:

```
{ Print " " [ Print [ Fail ] ] }     ⇦ Input
foo foo                              ⇦ Output
```

Example of propagating condition block:

```
{ Print " " [ Print < Fail > ] }     ⇦ Input
foo                                  ⇦ Ouput
```

Architecture

LWMDataButler

The desired capabilities of UTL are achieved using an object called a data butler. Data butlers are responsible for providing implementations for the macros encountered by the macro processor when executing template code. When the macro processor is invoked, a data butler interface[3] is supplied to it. Essentially, a data butler is a use of the Strategy pattern.[4]

---

[3] Interface defined in LWMDataButler.h

[4] See Gamma, Helm, Johnson Vlissides, Design Patterns

This accomplishes the first goal of the language design. By providing different data butlers to the macro processor, the grammar of UTL can be configured for the application and task at hand. Each data butler implementation can accept a set of keywords (macros) appropriate for its task and reject keywords that are not acceptable. Further, a given keyword can be implemented in different ways by different data butlers, allowing template code to be ignorant of the underlying implementation. This is useful in allowing multiple products to work from the same templates when their underlying data models differ.

The LWMDataButler interface has five entry points:

```
// Handle procedure macro with no parameters
virtual LWTBoolean SubstituteValue(
            const LWTString & MacroName,
            LWTString & ExpansionValue) = 0 ;

// Handle procedure macro with parameters
virtual LWTBoolean SubstituteValue(
            const LWTString & MacroName,
            LWMStringList & ParameterList,
            LWTString & ExpansionValue ) = 0 ;

// Handle iterator macro with no parameters
virtual LWTBoolean StartIteration(
            const LWTString & IterName ) = 0 ;

// Handle iterator macro with parameters
virtual LWTBoolean StartIteration(
            const LWTString & IterName,
            LWMStringList & ParameterList ) = 0 ;

// Handle iterator macro iteration
virtual LWTBoolean NextIteration(
            const LWTString & IterName ) = 0 ;
```

MCDataButler

The second design goal of UTL was to provide as much functionality as possible to the user and to allow them to publish their own code for reuse.

Since the metaphor of an UMA model deals with all data at a very abstract level—objects and properties—many grammatical elements of a template language can be written abstractly. These elements can be aggregated together to form the major portion of the language needed by any UMA-based product. The user could extend this base in the few areas that are product-specific. Ideally, the language should be extendable not only by the user, but by the end-user of the user's product. This is accomplished in our solution.

There are two basic problems that are addressed by the solution:

1. A mechanism is needed to share implementations. As new keywords are added, they must be published.

2. A mechanism is needed to control the context of a macro's execution. By this we refer to both the state of the UMA model at the time of execution, and to the specific object(s) or property(ies) the macro affects.

To meet these criteria, a base implementation of a data butler exists in UMA. This implementation, MCDataButlerI[5], provides mechanisms for defining the desired grammar incrementally and for managing contextual information in an UMA model.

This implementation is a superset of the LWMDataButler interface. Though the LWMDataButler interface is exposed, the user need not concern himself with it when implementing a data butler. The implementation in MCDataButlerI accomplishes all that has been found necessary to do in terms of implementing the LWMDataButler interface.

First, MCDataButlerI manages a registry of macro handlers. A macro handler is an object that knows how to perform a certain action, given a context. Conceptually, a macro handler is an example of a Command pattern.[6] The data butler maintains a dictionary of macro handlers keyed by macro name. When the macro processor requests the data butler to handle a macro (e.g. via SubstituteValue()), the data butler locates the appropriate macro handler in its dictionary and delegates control to it. The macro handler processes the request and returns control to the data butler who simply forwards the return value (if any) and the success state to the macro processor.

A macro handler exposes an interface that is used by the data butler for invocation:

```
virtual LWTBoolean Execute(
            MCDataButlerI * Butler,
            const LWTString & MacroName,
            LWMStringList * Parameters,
            LWTString * ExpansionValue) = 0;
```

MCDataButlerI exposes an implementation signature for registering new handlers:

```
void AddHandler(
            const LWTString & Macro,
            MCMacroHandlerBase * Handler);
```

In order to implement a grammar, a user can construct an implementation of an MCDataButlerI and implement all required macro handlers. Alternatively, since the macro handler registry is a member variable of a data butler instead of a singleton, the user can subclass an existing data butler. This provides the user with all macros known to the subclass, plus those they develop. An example of this is the OR•Compass Forward Engineering data butler; it subclasses MCDataButler. The latter implements a wide variety of generic macros for creating objects, reading properties, iterating, constructing variables, etc. The Forward Engineering data butler implements one or two macros associated only with Forward Engineering (such as a macro for determining the correct

[5] Implementation found in MCDataButlerInterface.h
[6] *Op cit.*, Design Patterns

execution command for a database) and ends up with a complete grammar for emitting DDL.

Second, MCDataButlerI manages a context stack. The context stack allows a macro handler to know what object in the UMA model is being referenced when the macro executes. For example, the `Property()` macro retrieves a named property from an object. The macro handler for `Property()` needs to know which object it should query.

MCDataButlerI maintains a stack of object references and exposes an interface for pushing and popping objects on this stack, as well as for querying the stack. Macro handlers can query this interface to find out the context of their operation.

```
MCObject * GetCurrentContext(void);
void PopCurrentContext(void);
void PushCurrentContext(MCObject * Object);
```

Additionally, MCDataButlerI exposes a facility that allows MCObject iterators to manage the context automatically. Macro handlers for iterator macros can instantiate an iterator proxy. This is an object that automatically maintains the context stack for the iterator. Upon instantiation, it pushes the first object being iterated onto the context stack. Each subsequent iteration causes a pop and a push to occur. When there are no more items to traverse, the stack is popped, leaving it in the state that existed when the iterator was encountered. These iterator proxies are maintained in a stack of their own in order to manage nested iterators.

Example—the following sample code shows how the Property macro would behave differently depending upon the context. Assume an entity named "MyTable1" that has an attribute "MyColumn1" and a second attribute "MyColumn2." Assume a second entity named "MyTable2" that has an attribute "MyColumn3." At the time this template code executes, we presume that the stack currently has the model on top due to some other code.

```
{                                                    ⇦ Input
   ForEach( "UOEntities" )
   begin
      Property( "UPName" )
      ForEach( "UPAttributes" )
      begin
         Property( "UPName" )
      end
   end
}

MyTable1 MyColumn1 MyColumn2 MyTable2 MyColumn3 ⇦ Output
```